Monitors
and Condition Synchronization

Abhik Roychoudhury
CS 3211
National University of Singapore

Reading material: Chapter 5 of Textbook.

In previous class

- Shared objects and mutual exclusion
  - Various threads trying to compete for access to shared object
  - Mutual exclusion ensured via Java synchronized statements and synchronized methods.
- Threads are Active Objects
- Shared Object being competed for is a passive object
  - Cannot change control flow on its own.
  - State change via (synchronized) method calls by threads.
  - Higher level concurrency concept
    - Java thread-safe shared object access essentially implements a **Monitor – a fundamental concurrency concept invented in 1974-75 by C.A.R. Hoare and Per Brinch Hansen.**

monitors & condition synchronization

Concepts: monitors:	encapsulated data + access procedures mutual exclusion + condition synchronization single access procedure active in the monitor nested monitors
Process equations:	guarded actions
Practice:	private data and synchronized methods (exclusion). wait(), notify() and notifyAll() for condition synch. single thread active in the monitor at a time

The concept of monitors

Brings in the concept of protected or private data.

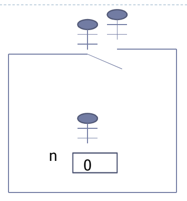
The protected data is accessed by several threads via operations
Protected data cannot be accessed without invoking the operations.
Each operation is executed atomically.

A monitor thus represents a passive object, whose operations are invoked by various active objects --- the threads.

A schematic monitor

```
monitor X{
    int n = 0;

    operation increment{
       int tmp;
       tmp = n ; n = tmp+1;
    }
}
```

Process p	Process q
X.increment	X.increment



Diagrammatic view of monitor X

The critical section code is encapsulated inside monitor operations, not replicated inside processes.

Extending monitors with conditions

Monitor operations may involve waiting on conditions (these are simple boolean expressions).

When such conditions become true, the waiting threads are notified (using wait, notify feature of Java).

Thus, each such condition has a waiting queue of blocked processes.
The schematic for conditional wait / notify are:

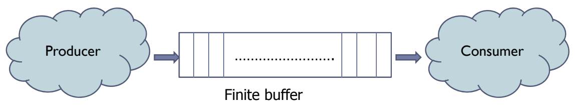
```
wait_on_cond(Cond)
    append p, the current proc. to queue for Cond
    p.state = blocked
    monitor.lock = released

signal_to_cond(Cond){
    if queue for Cond != empty{
        remove head of queue, let it be process x;
        x.state = ready
}
```

Implement semaphores using monitor

```
monitor Sem{
    integer s = 1  // initial value
    condition notZero

    operation acquire{
       if (s == 0)  wait_on_cond(notZero)
       s = s – 1;
    }

     operation release{
        s = s +1;
        signal_on_cond(notZero)
      }
}
```

```
USER PROCESS

while(1){
      non-critical section

      Sem.acquire();

       critical section

       Sem.release();
}
```

Producer consumer problem



Producer: blocks if buffer is full.
Consumer: blocks if buffer is empty.

Schematic Producer-Consumer

```
monitor PC{
      buffer = empty;
      condition notFull, notEmpty;

      operation produce(v){
           if buffer is full{
                 wait_on_cond(notFull)
           }
          add v to tail of buffer;
          signal_to_cond(notEmpty)
       }
```

```
operation consume(){
     if buffer is empty{
           wait_on_cond(notEmpty);
     }
     remove w from head of buffer;
     signal_to_cond(notFull);
     return w;
}
```

Producer	Consumer

while (1){
    d = get_new_item;
    PC.produce(d);
}
	while (1){
	    d = PC.consume();
	    put_item(d);
	}

Monitors in Java

Not a default construct.
Need to be programmed as a new class with private data (the data being protected) and synchronized methods.

Blocking of processes is supported by	wait()
Unblocking of processes is supported by	notify(), notifyAll()

wait() can throw exceptions, so we will add code to catch them.

Producer-consumer in Java

```
class PCMonitor{
      final int N = 5;
      int Oldest = 0, Newest = 0;
      volatile int Count = 0;
      int Buffer[] = new int[N];

synchronized void produce(int v){
    while (Count == N) try{ wait();} catch(InterruptedException e) {}
    Buffer[Newest] =V;
    Newest = (Newest + 1) %N;
    Count++;  notifyAll();
}

synchronized int consume(){
     int tmp;
     while (Count == 0) try{ wait();} catch(InterruptedException e) {}
     tmp = Buffer[Oldest]; Oldest = (Oldest + 1) % N;
     Count--; notifyAll();
      return tmp;
}
```

Readers-Writers Problem

Several processes accessing a common resource.

Accessing processes grouped into two categories.

Readers: do not exclude other readers, exclude writers.
Writers: exclude all other processes while accessing.

How to give a solution using monitors?

Schematic Readers-Writers

```
monitor RW{
    int readers=0, writers=0;
    condition OKtoRead, OKtoWrite;

    operation StartRead{
        if writers !=0 ∨ not empty(OKtoWrite){ wait_on_cond(OKtoRead);}
        readers++;  signal_to_cond(OKtoRead);
    }
    operation EndRead{
        readers--;  if readers == 0 { signal_to_cond(OKtoWrite);}
    }
    operation StartWrite{
        if writers!=0 ∨ readers != 0 { wait_on_cond(OKtoWrite); }
       writers++;
    }
    operation EndWrite{
        writers--;
        if empty(OKtoRead){signal_to_cond(OKtoWrite);}
       else { signal_to_cond(OKtoRead); }
    }
}
```

This is schematic code – it does not reflect the solution in Java.

Correctness of Readers-Writers

R = Number of readers
W = Number of writers

Invariant property

$(R > 0 \Rightarrow W == 0) \wedge (W \leq 1) \wedge (W == 1 \Rightarrow R == 0)$

Prove that it is preserved by each of the operations of the RW monitor.

Doing it in Java

Java has no mechanism for waiting on a specific condition.

We can call the wait() method of any Java object, which suspends the current thread. The thread is said to be "waiting on" the given object.

Another thread calls the notify() method of the same Java object. This "wakes up" one of the threads waiting on that object.

```
synchronized method1(){                      synchronized  method2(){
    while (x==0) wait();                          while (y==0) wait();
}                                            }

synchronized method3(…){
    if (…) x = 1 else y = 1;
    notifyAll();
}
```

If wrong process is notified it will return itself to the set of waiting processes.

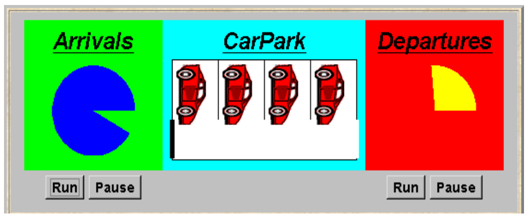
So Far …

A basic idea of what monitor is
-- protected data
-- atomic access via methods
Basic Examples to show usage of monitors
-- Producer-consumer
-- Readers-writers

Now

More advanced / detailed examples with monitors.

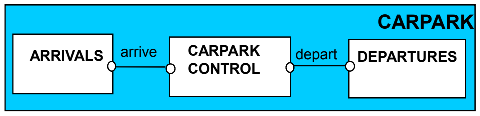
5.1 Condition synchronization



A controller is required for a carpark, which only permits cars to enter when the carpark is not full and does not permit cars to leave when there are no cars in the carpark. Car arrival and departure are simulated by separate threads.

carpark model

- Events or actions of interest?
  arrive and depart
- Identify processes.
  arrivals, departures and carpark control
- Define each process and interactions (structure).



carpark model

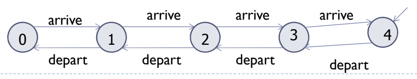
```
CARPARKCONTROL(N=4)  = SPACES[N],
SPACES[i:0..N]  =  (when(i>0) arrive->SPACES[i-1]
                   |when(i<N) depart->SPACES[i+1]
                   ).

ARRIVALS    = (arrive->ARRIVALS).
DEPARTURES = (depart->DEPARTURES).

||CARPARK =
        (ARRIVALS||CARPARKCONTROL(4)||DEPARTURES).
```

Guarded actions are used to control **arrive** and **depart**.

LTS?

Carpark LTS

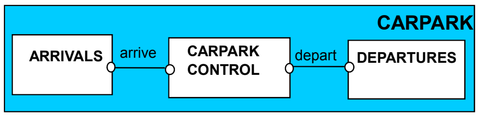
```
CARPARKCONTROL(N=4)  = SPACES[N],
SPACES[i:0..N]  =  (when(i>0) arrive->SPACES[i-1]
                   |when(i<N) depart->SPACES[i+1]
                   ).

ARRIVALS    = (arrive->ARRIVALS).
DEPARTURES = (depart->DEPARTURES).

CARPARK =
        (ARRIVALS||CARPARKCONTROL(4)||DEPARTURES).
```



carpark program

- Model - all entities are processes interacting by actions
- Program - need to identify threads and monitors
  - thread - **active** entity which initiates (output) actions
  - monitor - **passive** entity which responds to (input) actions.

For the carpark?



carpark program

Arrivals and **Departures** implement **Runnable**, **CarParkControl** provides the control (condition synchronization).

Instances of these are created by the **start()** method of the **CarPark** applet

```
public void start() {
  CarParkControl c =
     new DisplayCarPark(carDisplay,Places);
     arrivals.start(new Arrivals(c));
     departures.start(new Departures(c));
}
```

carpark program - `Arrivals` and `Departures` threads

```
class Arrivals implements Runnable {
  CarParkControl carpark;

  Arrivals(CarParkControl c) {carpark = c;}

  public void run() {
    try {
      while(true) {
        ThreadPanel.rotate(330);
        carpark.arrive();
        ThreadPanel.rotate(30);
      }
    } catch (InterruptedException e){}
  }
}
```

Similarly **Departures** which calls **carpark.depart()**.

How do we implement the control of **CarParkControl**?

Carpark program - `CarParkControl` monitor

```
class CarParkControl {
  protected int spaces;
  protected int capacity;

  CarParkControl(int capacity)
    {spaces = capacity;}

  synchronized void arrive() {
    …  --spaces; …
  }

  synchronized void depart() {
    … ++spaces; …
  }
}
```

mutual exclusion by synch methods

condition synchronization?

block if full? (spaces==0)

block if empty? (spaces==N)

condition synchronization in Java

Java provides a thread **wait queue** per monitor (actually per object) with the following methods:

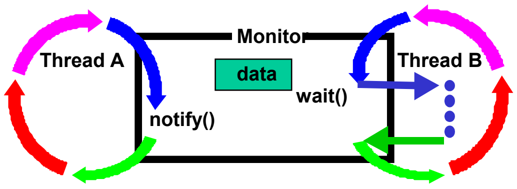
`public final void notify()` Wakes up a single thread that is waiting on this object's queue.

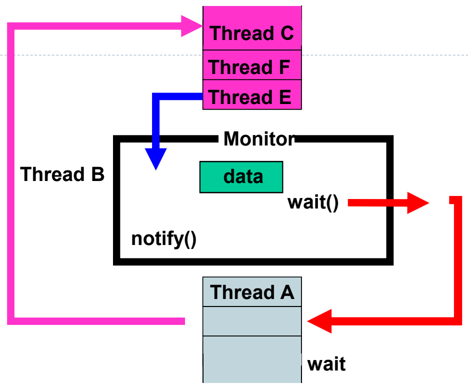
`public final void notifyAll()`
Wakes up all threads that are waiting on this object's queue.

```
public final void wait()
    throws InterruptedException
```
Waits to be notified by another thread. The waiting thread releases the synchronization lock associated with the monitor. When notified, the thread must wait to reacquire the monitor before resuming execution.

condition synchronization in Java

We refer to a thread ***entering*** a monitor when it acquires the mutual exclusion lock associated with the monitor and ***exiting*** the monitor when it releases the lock.
Wait() - causes the thread to exit the monitor, permitting other threads to enter the monitor.





condition synchronization in Java

```
FSP:   when cond act -> NEWSTAT
```

```
Java:  public synchronized void act()
              throws InterruptedException
         {
          while (!cond) wait();
          // modify monitor data
          notifyAll()
         }
```

The `while` loop is necessary to retest the condition *cond* to ensure that *cond* is indeed satisfied when it re-enters the monitor.

`notifyall()` is necessary to awaken other thread(s) that may be waiting to enter the monitor now that the monitor data has been changed.

`CarParkControl` - condition synchronization

```
class CarParkControl {
  protected int spaces;
  protected int capacity;

  CarParkControl(int capacity)
    {capacity = spaces = n;}

  synchronized void arrive() throws InterruptedException {
    while (spaces==0) wait();
    --spaces;
    notify();
  }

  synchronized void depart() throws InterruptedException {
    while (spaces==capacity) wait();
    ++spaces;
    notify();
  }
}
```

when(i>0) arrive ->SPACES[i-1]

when(i<N) depart ->SPACES[i+1]

Why is it safe to use `notify()` here rather than `notifyAll()`?

Monitors are passive

Active entities (that initiate actions) are implemented as **threads**.
Passive entities (that respond to actions) are implemented as **monitors**.

Each guarded action in the model of a monitor is implemented as a `synchronized` method which uses a while loop and `wait()` to implement the guard. The while loop condition is the negation of the model guard condition.

Changes in the state of the monitor are signaled to waiting threads using `notify()` or `notifyAll()`.

5.2 Semaphores

Semaphores are widely used for dealing with inter-process synchronization in operating systems. Semaphore ***s*** is an integer variable that can take only non-negative values.

The only operations permitted on ***s*** are *up(s)* and *down(s)*.

down(s): when s>0 decrement(s)

up(s): increment(s)

Does this mean there will be busy waiting?

modeling semaphores

To ensure analyzability, we only model semaphores that take a finite range of values. If this range is exceeded then we regard this as an ERROR. N is the initial value.

```
const Max = 3
range Int = 0..Max

SEMAPHORE(N)  = SEMA[N],
SEMA[v:Int]     = (up->SEMA[v+1]
                  |when(v>0) down->SEMA[v-1]
                  ),
SEMA[Max+1]     = ERROR.
```

LTS?

semaphore demo - model

Three processes `p[1..3]` use a shared semaphore `mutex` to ensure mutually exclusive access (action `critical`) to some resource.

```
LOOP = (mutex.down->critical->mutex.up->LOOP).
||SEMADEMO = (p[1..3]:LOOP
           ||{p[1..3]}::mutex:SEMAPHORE(1)).

 SEMAPHORE(N)  = SEMA[N],
 SEMA[v:Int]     = (up->SEMA[v+1]
                  |when(v>0) down->SEMA[v-1]
                  ),
 SEMA[Max+1]     = ERROR.
```

Semaphore

For mutual exclusion, the semaphore initial value is 1. ***Why?***

Is the `ERROR` *state reachable for* `SEMADEMO`***?***

Is a binary semaphore sufficient (i.e. `Max=1`*)* ***?***

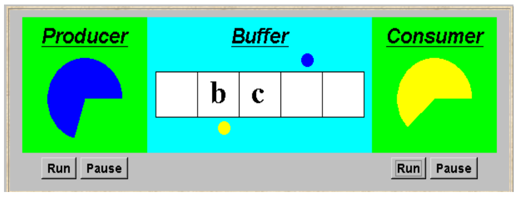
semaphores in Java

Semaphores are passive objects, therefore implemented as **monitors**.

(In practice, semaphores are a low-level mechanism often used in implementing the higher-level monitor construct.)

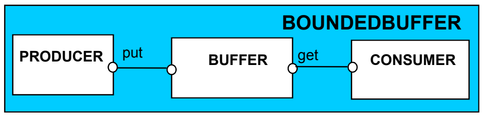
```
public class Semaphore {
  private int value;

  public Semaphore (int initial)
     {value = initial;}

  synchronized public void up() {
     ++value;
     notify();
  }

  synchronized public void down()
      throws InterruptedException {
    while (value== 0) wait();
    --value;
  }
}
```

5.3 Bounded Buffer



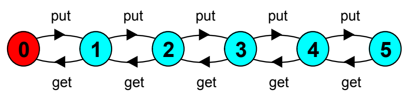
A bounded buffer consists of a fixed number of slots. Items are put into the buffer by a *producer* process and removed by a *consumer* process. It can be used to smooth out transfer rates between the *producer* and *consumer.*

bounded buffer - a data-independent model



The behaviour of BOUNDEDBUFFER is independent of the actual data values, and so can be modelled in a data-independent manner.

LTS:



bounded buffer - a data-independent model

```
BUFFER(N=5)  = COUNT[0],
COUNT[i:0..N]
     = (when  (i<N)  put->COUNT[i+1]
       |when  (i>0)  get->COUNT[i-1]
       ).

PRODUCER = (put->PRODUCER).
CONSUMER = (get->CONSUMER).

||BOUNDEDBUFFER =
(PRODUCER||BUFFER(5)||CONSUMER).
```

bounded buffer program - buffer monitor

```
public interface Buffer {…}

class BufferImpl implements Buffer {
     …
 public synchronized void put(Object o)
           throws InterruptedException {
    while (count==size) wait();
    buf[in] = o; ++count; in=(in+1)%size;
    notify();
  }
  public synchronized Object get()
           throws InterruptedException {
    while (count==0) wait();
    Object o =buf[out];
    buf[out]=null; --count; out=(out+1)%size;
    notify();
    return (o);
    }

}
```

bounded buffer program - producer process

```
class Producer implements Runnable {
  Buffer<Character> buf;
  String alphabet= "abcdefghijklmnopqrstuvwxyz";

  Producer(Buffer<Character> b) {buf = b;}

  public void run() {
    try {
      int ai = 0;
      while(true) {
        ThreadPanel.rotate(12);
        buf.put(new Character(alphabet.charAt(ai)));
        ai=(ai+1) % alphabet.length();
        ThreadPanel.rotate(348);
      }
    } catch (InterruptedException e){}
  }
}
```

Similarly **Consumer** which calls **buf.get()**.

bounded buffer program - consumer process

```
class Producer implements Runnable {
  Buffer<Character> buf;

  Consumer(Buffer<Character> b) {buf = b;}

  public void run() {
    try {
      while(true) {
        ThreadPanel.rotate(180);
        Character c = buf.get();
        ThreadPanel.rotate(348);
      }
    } catch (InterruptedException e){}
  }
}
```

Similarly **Consumer** which calls **buf.get()**.

5.4 Nested Monitors

Suppose that, in place of using the ***count*** variable and condition synchronization directly, we instead use two semaphores ***full*** and ***empty*** to reflect the state of the buffer.

```
class SemaBuffer implements Buffer {
  …
  Semaphore full;   //counts number of items
  Semaphore empty;  //counts number of spaces

  SemaBuffer(int size) {
    this.size = size; buf = new Object[size];
    full = new Semaphore(0);
    empty= new Semaphore(size);
  }
…
}
```

nested monitors - bounded buffer program

```
synchronized public void put(Object o)
              throws InterruptedException {
   empty.down();
   buf[in] = o;
   ++count; in=(in+1)%size;
   full.up();
 }
  synchronized public Object get()
               throws InterruptedException{
   full.down();
   Object o =buf[out]; buf[out]=null;
   --count; out=(out+1)%size;
   empty.up();
   return (o);
 }
```

Does this behave as desired?

empty is decremented during a **put** operation, which is blocked if ***empty*** is zero; ***full*** is decremented by a **get** operation, which is blocked if ***full*** is zero.

Deadlock scenario

1. Initially buffer does not contain anything
2. Consumer wants to execute get() operation
   // This should block since buffer is empty.
3. Inside get() --- full.down() is executed
   // since full is 0 --- this causes Java wait()
   // the execution of wait() releases the lock for full
   // the execution of wait() does not release the lock for SemaBuffer
4. Producer cannot acquire the lock for Semabuffer
5. Consumer also keeps on waiting since the producer does not get a chance to insert anything in the buffer.

nested monitors - bounded buffer program

```
synchronized public void put(Object o)
              throws InterruptedException {
   empty.down();
   buf[in] = o;
   ++count; in=(in+1)%size;
   full.up();
 }
  synchronized public Object get()
               throws InterruptedException{
   full.down();
   Object o =buf[out]; buf[out]=null;
   --count; out=(out+1)%size;
   empty.up();
   return (o);
 }
```

Does this behave as desired?

```
synchronized public void down()
     throws InterruptedException {
   while (value== 0) wait();
   --value;
 }
```

Going through it one more time

Initially buffer does not contain anything,
Integer protected by semaphore *full* is 0,
And integer protected by semaphore *empty* is non-zero

Consumer executes get()

Inside get(), the first line is full.down()

Inside down, the first line is
while (value == 0) wait()
// value is the integer protected by the semaphore monitor

Since *full* is 0, wait() is executed
Since wait() is encountered in a method for the *full* semaphore – it releases the lock for *full*

The lock for the buffer whose get() called full.down() is not released!!

nested monitors - revised bounded buffer program

The only way to avoid it in Java is by careful design. In this example, the deadlock can be removed by ensuring that the monitor lock for the buffer is not acquired until ***after*** semaphores are decremented.

```
public void put(Object o)
          throws InterruptedException {
   empty.down();
   synchronized(this){
     buf[in] = o; ++count; in=(in+1)%size;
   }
   full.up();
 }
```

nested monitors - revised bounded buffer model

```
BUFFER =  (put -> BUFFER
          |get -> BUFFER
          ).

PRODUCER =(empty.down->put->full.up->PRODUCER).
CONSUMER =(full.down->get->empty.up->CONSUMER).
```

The semaphore actions have been moved to the producer and consumer. This is exactly as in the implementation where the semaphore actions are ***outside*** the monitor .

Does this behave as desired?

Minimized LTS?

5.5 Monitor invariants

An **invariant** for a monitor is an assertion concerning the variables it encapsulates. This assertion must hold whenever there is no thread executing inside the monitor i.e. on thread **entry** to and **exit** from a monitor. They are useful for us to gain understanding of a given monitor.

CarParkControl Invariant:		$0 \le spaces \le N$
Semaphore Invariant:		$0 \le value$
Buffer Invariant:		$0 \le count \le size$
	and	$0 \le in < size$
	and	$0 \le out < size$
	and	$in = (out + count)$ **modulo** $size$

Entry to monitor: acquisition of the lock for the monitor.

Exit from monitor: wait(), or exit from synchronized method call

Summary

- Concepts
  - **monitors: encapsulated data + access procedures**
    mutual exclusion + condition synchronization
  - **nested monitors**
- Process Equations
  - **guarded actions**
- Practice (Java)
  - **private data and synchronized methods in Java**
  - **wait(), notify() and notifyAll() for condition synchronization**
  - **single thread active in the monitor at a time**